vae

April 16, 2024

We would like to acknowledge University of Michigan's EECS 498-007/598-005 on which we based the development of this project.

1 Variational Autoencoder

In this notebook, you will implement a variational autoencoder and a conditional variational autoencoder with slightly different architectures and apply them to the popular MNIST handwritten dataset. Recall from C147/C247, an autoencoder seeks to learn a latent representation of our training images by using unlabeled data and learning to reconstruct its inputs. The *variational autoencoder* extends this model by adding a probabilistic spin to the encoder and decoder, allowing us to sample from the learned distribution of the latent space to generate new images at inference time.

1.1 Setup Code

Before getting started, we need to run some boilerplate code to set up our environment. You'll need to rerun this setup code each time you start the notebook.

First, run this cell that loads the autoreload extension. This allows us to edit .py source files and re-import them into the notebook for a seamless editing and debugging experience.

[1]:
```
%load_ext autoreload
%autoreload 2
```

1.1.1 Google Colab Setup

Next we need to run a few commands to set up our environment on Google Colab. If you are running this notebook on a local machine you can skip this section.

Run the following cell to mount your Google Drive. Follow the link and sign in to your Google account (the same account you used to store this notebook!) and copy the authorization code into the text box that appears below.

[2]:
```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Now recall the path in your Google Drive where you uploaded this notebook and fill it in below. If everything is working correctly then running the folowing cell should print the filenames from the assignment:

```
['vae.ipynb', 'nndl2', 'vae.py']
```

[3]:
```
import os

# TODO: Fill in the Google Drive path where you uploaded the assignment
# Example: '239AS.3/project1/vae'
GOOGLE_DRIVE_PATH_AFTER_MYDRIVE = '24S_239AS3_NNDL2/Project1/vae'
GOOGLE_DRIVE_PATH = os.path.join('drive', 'My Drive',␣
  ↪GOOGLE_DRIVE_PATH_AFTER_MYDRIVE)
print(os.listdir(GOOGLE_DRIVE_PATH))
```

```
['vae.py', '.DS_Store', 'nndl2', 'vae.ipynb']
```

Once you have successfully mounted your Google Drive and located the path to this assignment, run the following cell to allow us to import from the .py files of this assignment. If it works correctly, it should print the message:

```
Hello from vae.py!
Hello from helper.py!
```

[4]:
```
import sys
sys.path.append(GOOGLE_DRIVE_PATH)

from vae import hello_vae
hello_vae()

from nndl2.helper import hello_helper
hello_helper()
```

```
Hello from vae.py!
Hello from helper.py!
```

Load several useful packages that are used in this notebook:

[5]:
```
from nndl2.grad import rel_error
from nndl2.utils import reset_seed
import math
import torch
import torch.nn as nn
import torch.nn.functional as F
from torch.nn import init
import torchvision
import torchvision.transforms as T
import torch.optim as optim
from torch.utils.data import DataLoader
from torch.utils.data import sampler
```

```
import torchvision.datasets as dset

import matplotlib.pyplot as plt
%matplotlib inline


# for plotting
plt.rcParams['figure.figsize'] = (10.0, 8.0) # set default size of plots
plt.rcParams['font.size'] = 16
plt.rcParams['image.interpolation'] = 'nearest'
plt.rcParams['image.cmap'] = 'gray'
```

We will use GPUs to accelerate our computation in this notebook. Run the following to make sure GPUs are enabled:

[6]:
```
if torch.cuda.is_available():
  print('Good to go!')
else:
  print('Please set GPU via the downward triangle in the top right corner.')
```

```
Good to go!
```

1.2 Load MNIST Dataset

VAEs are notoriously finicky with hyperparameters, and also require many training epochs. In order to make this assignment approachable, we will be working on the MNIST dataset, which is 60,000 training and 10,000 test images. Each picture contains a centered image of white digit on black background (0 through 9). This was one of the first datasets used to train convolutional neural networks and it is fairly easy – a standard CNN model can easily exceed 99% accuracy.

To simplify our code here, we will use the PyTorch MNIST wrapper, which downloads and loads the MNIST dataset. See the documentation for more information about the interface. The default parameters will take 5,000 of the training examples and place them into a validation dataset. The data will be saved into a folder called `MNIST`.

[7]:
```
%cd /content/drive/My\ Drive/$GOOGLE_DRIVE_PATH_AFTER_MYDRIVE

batch_size = 128

mnist_train = dset.MNIST('./nndl2', train=True, download=True,
                           transform=T.ToTensor())
loader_train = DataLoader(mnist_train, batch_size=batch_size,
                          shuffle=True, drop_last=True, num_workers=2)
```

```
/content/drive/My Drive/24S_239AS3_NNDL2/Project1/vae
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-images-idx3-ubyte.gz to
./nndl2/MNIST/raw/train-images-idx3-ubyte.gz
```

```
100%|          | 9912422/9912422 [00:00<00:00, 188718872.67it/s]
```

```
Extracting ./nndl2/MNIST/raw/train-images-idx3-ubyte.gz to ./nndl2/MNIST/raw
```

```
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/train-labels-idx1-ubyte.gz to
./nndl2/MNIST/raw/train-labels-idx1-ubyte.gz

100%|      | 28881/28881 [00:00<00:00, 135045366.58it/s]

Extracting ./nndl2/MNIST/raw/train-labels-idx1-ubyte.gz to ./nndl2/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-images-idx3-ubyte.gz to
./nndl2/MNIST/raw/t10k-images-idx3-ubyte.gz

100%|      | 1648877/1648877 [00:00<00:00, 146179353.57it/s]

Extracting ./nndl2/MNIST/raw/t10k-images-idx3-ubyte.gz to ./nndl2/MNIST/raw

Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz
Downloading http://yann.lecun.com/exdb/mnist/t10k-labels-idx1-ubyte.gz to
./nndl2/MNIST/raw/t10k-labels-idx1-ubyte.gz

100%|      | 4542/4542 [00:00<00:00, 18640439.11it/s]

Extracting ./nndl2/MNIST/raw/t10k-labels-idx1-ubyte.gz to ./nndl2/MNIST/raw
```

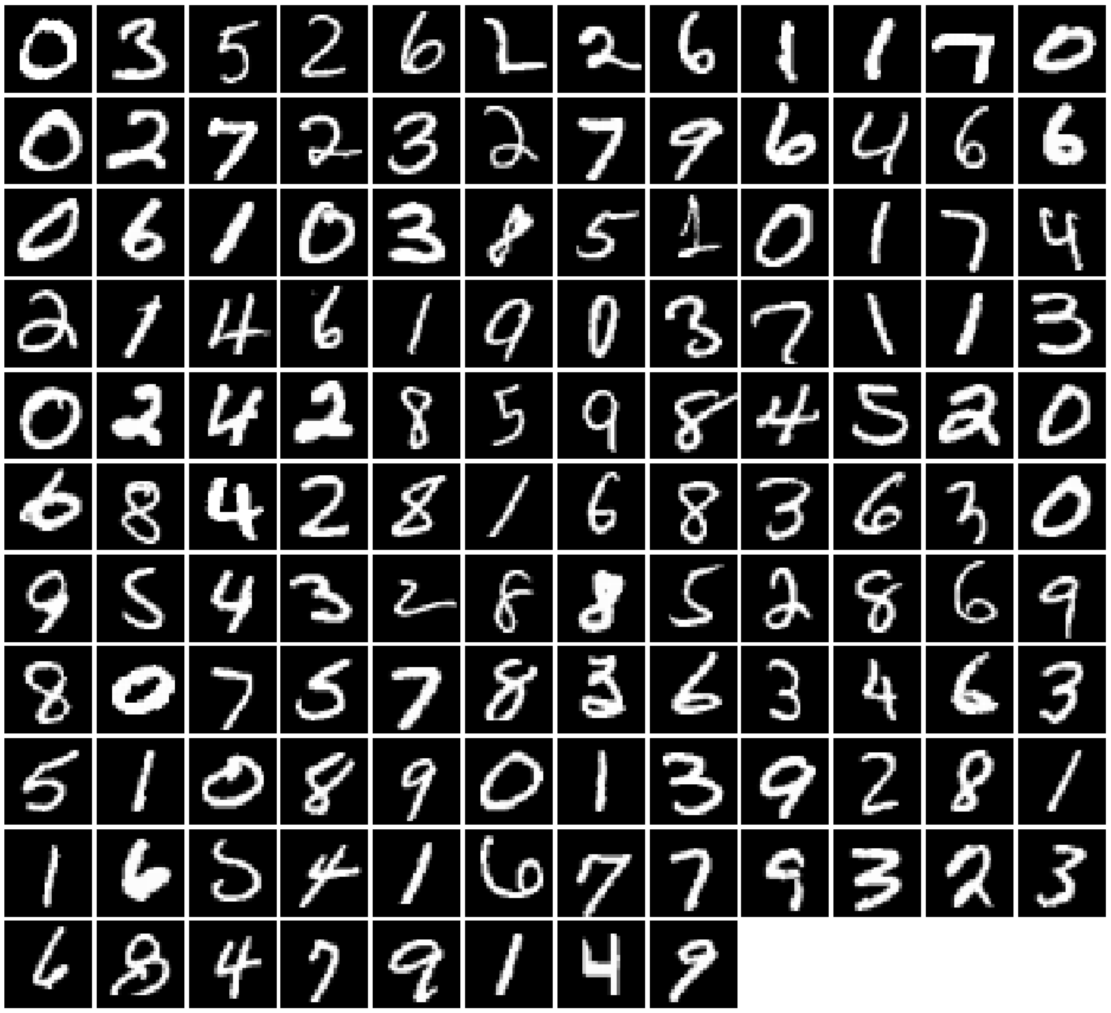
1.3 Visualize dataset

It is always a good idea to look at examples from the dataset before working with it. Let's visualize the digits in the MNIST dataset. We have defined the function `show_images` in `helper.py` that we call to visualize the images.

[8]:
```python
from nndl2.helper import show_images

imgs = next(iter(loader_train))[0].view(batch_size, 784)
show_images(imgs)
```

```
/usr/lib/python3.10/multiprocessing/popen_fork.py:66: RuntimeWarning: os.fork()
was called. os.fork() is incompatible with multithreaded code, and JAX is
multithreaded, so this will likely lead to a deadlock.
  self.pid = os.fork()
```



2 Fully Connected VAE

Our first VAE implementation will consist solely of fully connected layers. We'll take the `1 x 28 x 28` shape of our input and flatten the features to create an input dimension size of 784. In this section you'll define the Encoder and Decoder models in the VAE class of `vae.py` and implement the reparametrization trick, forward pass, and loss function to train your first VAE.

2.1 FC-VAE Encoder (4 points)

Now lets start building our fully-connected VAE network. We'll start with the encoder, which will take our images as input (after flattening C,H,W to D shape) and pass them through a three Linear+ReLU layers. We'll use this hidden dimension representation to predict both the posterior mu and posterior log-variance using two separate linear layers (both shape (N,Z)).

Note that we are calling this the 'logvar' layer because we'll use the log-variance (instead of variance or standard deviation) to stabilize training. This will specifically matter more when you compute reparametrization and the loss function later.

Define `hidden_dim=400`*,* `encoder`*,* `mu_layer`*, and* `logvar_layer` *in the initialization of the VAE class in* `vae.py`*. Use nn.Sequential to define the encoder, and separate Linear layers for the mu and logvar layers. Architecture for the encoder is described below:*

- `Flatten` (Hint: nn.Flatten)
- Fully connected layer with input size `input_size` and output size `hidden_dim`
- `ReLU`
- Fully connected layer with input_size `hidden_dim` and output size `hidden_dim`
- `ReLU`
- Fully connected layer with input_size `hidden_dim` and output size `hidden_dim`
- `ReLU`

2.2 FC-VAE Decoder (1 point)

We'll now define the decoder, which will take the latent space representation and generate a reconstructed image. The architecture is as follows:

- Fully connected layer with input size `latent_size` and output size `hidden_dim`
- `ReLU`
- Fully connected layer with input_size `hidden_dim` and output size `hidden_dim`
- `ReLU`
- Fully connected layer with input_size `hidden_dim` and output size `hidden_dim`
- `ReLU`
- Fully connected layer with input_size `hidden_dim` and output size `input_size`
- `Sigmoid`
- `Unflatten` (nn.Unflatten)

Define a `decoder` *in the initialization of the VAE class in* `vae.py`*. Like the encoding step, use* `nn.Sequential`

2.3 Reparametrization (2 points)

Now we'll apply a reparametrization trick in order to estimate the posterior z during our forward pass, given the μ and σ^2 estimated by the encoder. A simple way to do this could be to simply generate a normal distribution centered at our μ and having a std corresponding to our σ^2. However, we would have to backpropogate through this random sampling that is not differentiable. Instead, we sample initial random data ϵ from a fixed distrubtion, and compute z as a function of (ϵ, σ^2, μ). Specifically:

$z = \mu + \sigma\epsilon$

We can easily find the partial derivatives w.r.t μ and σ^2 and backpropagate through z. If $\epsilon = \mathcal{N}(0, 1)$, then its easy to verify that the result of our forward pass calculation will be a distribution centered at μ with variance σ^2.

Implement `reparametrization` in `vae.py` and verify your mean and std error are at or less than `1e-4`.

[9]:
```python
reset_seed(0)
from vae import reparametrize
```

```
latent_size = 15
size = (1, latent_size)
mu = torch.zeros(size)
logvar = torch.ones(size)

z = reparametrize(mu, logvar)

expected_mean = torch.FloatTensor([-0.4363])
expected_std = torch.FloatTensor([1.6860])
z_mean = torch.mean(z, dim=-1)
z_std = torch.std(z, dim=-1)
assert z.size() == size

print('Mean Error', rel_error(z_mean, expected_mean))
print('Std Error', rel_error(z_std, expected_std))
```

```
Mean Error 5.639056398351415e-05
Std Error 7.1412955526273885e-06
```

2.4 FC-VAE Forward (1 point)

Complete the VAE class by writing the forward pass. The forward pass should pass the input image through the encoder to calculate the estimation of mu and logvar, reparametrize to estimate the latent space z, and finally pass z into the decoder to generate an image.

2.5 Loss Function (1 point)

Before we're able to train our final model, we'll need to define our loss function. As seen below, the loss function for VAEs contains two terms: A reconstruction loss term (left) and KL divergence term (right).

$-E_{Z\ q_\phi(z|x)}[log p_\theta(x|z)] + D_{KL}(q_\phi(z|x), p(z)))$

Note that this is the negative of the variational lowerbound shown in lecture–this ensures that when we are minimizing this loss term, we're maximizing the variational lowerbound. The reconstruction loss term can be computed by simply using the binary cross entropy loss between the original input pixels and the output pixels of our decoder (Hint: `nn.functional.binary_cross_entropy`). The KL divergence term works to force the latent space distribution to be close to a prior distribution (we're using a standard normal gaussian as our prior).

To help you out, we've derived an unvectorized form of the KL divergence term for you. Suppose that $q_\phi(z|x)$ is a Z-dimensional diagonal Gaussian with mean $\mu_{z|x}$ of shape $(Z,)$ and standard deviation $\sigma_{z|x}$ of shape $(Z,)$, and that $p(z)$ is a Z-dimensional Gaussian with zero mean and unit variance. Then we can write the KL divergence term as:

$D_{KL}(q_\phi(z|x), p(z))) = -\frac{1}{2}\sum_{j=1}^{J}(1 + log(\sigma_{z|x}^2)_j - (\mu_{z|x})_j^2 - (\sigma_{z|x})_j^2)$

It's up to you to implement a vectorized version of this loss that also operates on minibatches. You should average the loss across samples in the minibatch.

Implement `loss_function` in `vae.py` and verify your implementation below. Your relative error should be less than or equal to `1e-5`

[10]:
```
from vae import loss_function
size = (1,15)

image = torch.sigmoid(torch.FloatTensor([[2,5], [6,7]]).unsqueeze(0).
  ↪unsqueeze(0))
image_hat = torch.sigmoid(torch.FloatTensor([[1,10], [9,3]]).unsqueeze(0).
  ↪unsqueeze(0))

expected_out = torch.tensor(8.5079)
mu, logvar = torch.ones(size), torch.zeros(size)
out = loss_function(image, image_hat, mu, logvar)
print('Loss error', rel_error(expected_out,out))
```

```
Loss error 2.1297676389877955e-06
```

2.6 Train a model

Now that we have our VAE defined and loss function ready, lets train our model! Our training script is provided in `nndl2/helper.py`, and we have pre-defined an Adam optimizer, learning rate, and # of epochs for you to use.

Training for 10 epochs should take ~2 minutes and your loss should be less than 120.

[11]:
```
num_epochs = 10
latent_size = 15
from vae import VAE
from nndl2.helper import train_vae
input_size = 28*28
device = 'cuda'
vae_model = VAE(input_size, latent_size=latent_size)
vae_model.cuda()
for epoch in range(0, num_epochs):
  train_vae(epoch, vae_model, loader_train)
```

```
Train Epoch: 0  Loss: 155.775269
Train Epoch: 1  Loss: 141.299377
Train Epoch: 2  Loss: 125.814720
Train Epoch: 3  Loss: 121.087906
Train Epoch: 4  Loss: 121.011955
Train Epoch: 5  Loss: 117.449791
Train Epoch: 6  Loss: 113.615326
Train Epoch: 7  Loss: 114.995316
Train Epoch: 8  Loss: 112.340248
Train Epoch: 9  Loss: 108.839287
```

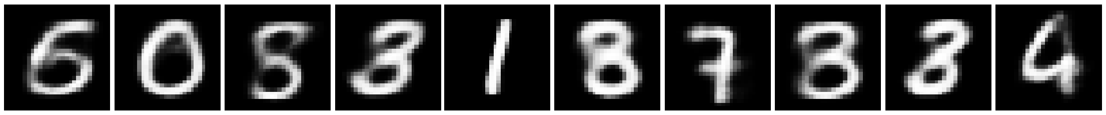
2.7 Visualize results

After training our VAE network, we're able to take advantage of its power to generate new training examples. This process simply involves the decoder: we intialize some random distribution for our latent spaces z, and generate new examples by passing these latent space into the decoder.

Run the cell below to generate new images! You should be able to visually recognize many of the digits, although some may be a bit blurry or badly formed. Our next model will see improvement in these results.

[12]:
```python
z = torch.randn(10, latent_size).to(device='cuda')
import matplotlib.gridspec as gridspec
vae_model.eval()
samples = vae_model.decoder(z).data.cpu().numpy()

fig = plt.figure(figsize=(10, 1))
gspec = gridspec.GridSpec(1, 10)
gspec.update(wspace=0.05, hspace=0.05)
for i, sample in enumerate(samples):
  ax = plt.subplot(gspec[i])
  plt.axis('off')
  ax.set_xticklabels([])
  ax.set_yticklabels([])
  ax.set_aspect('equal')
  plt.imshow(sample.reshape(28,28), cmap='Greys_r')
```



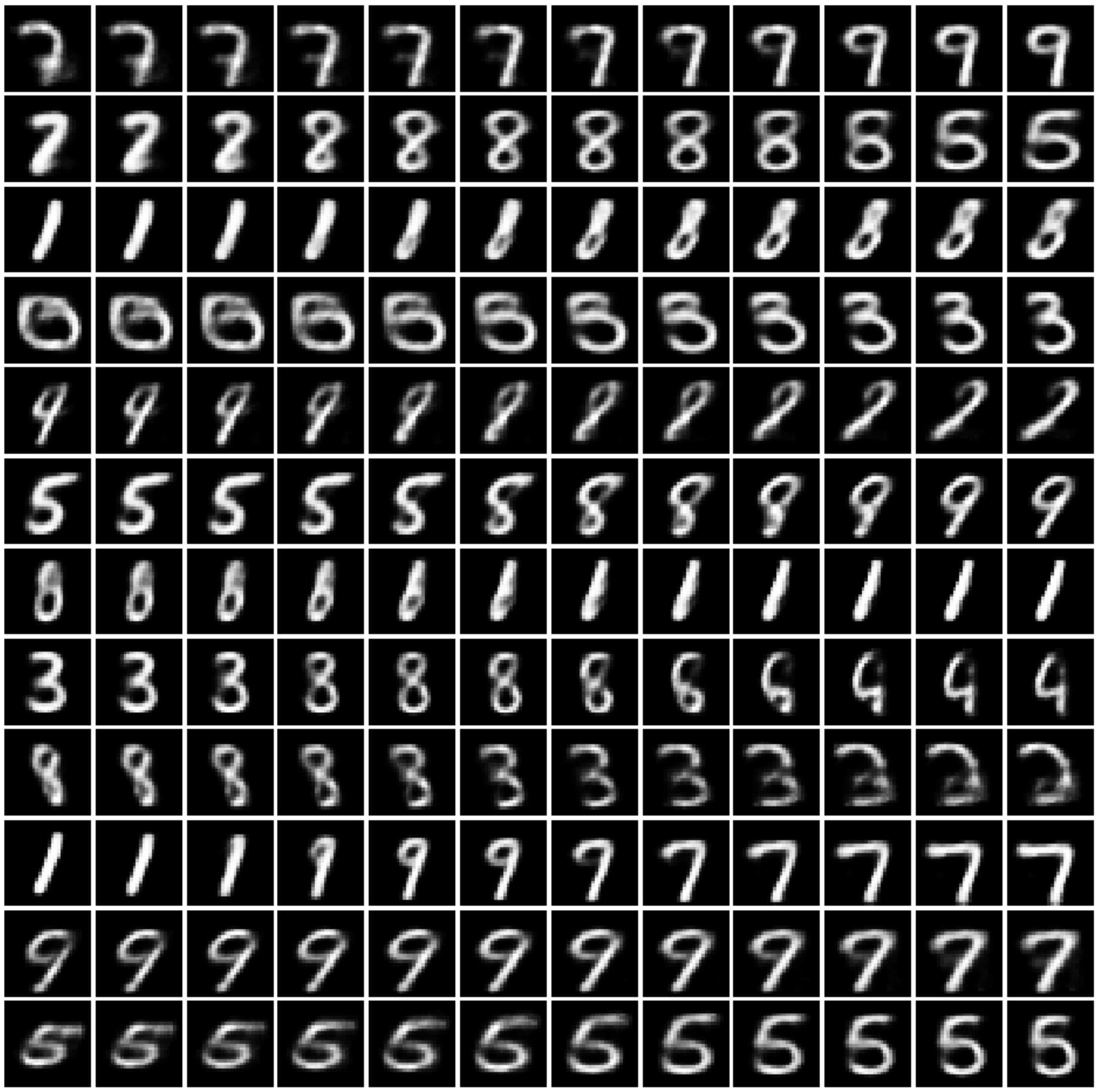
2.8 Latent Space Interpolation

As a final visual test of our trained VAE model, we can perform interpolation in latent space. We generate random latent vectors z_0 and z_1, and linearly interplate between them; we run each interpolated vector through the trained generator to produce an image.

Each row of the figure below interpolates between two random vectors. For the most part the model should exhibit smooth transitions along each row, demonstrating that the model has learned something nontrivial about the underlying spatial structure of the digits it is modeling.

[13]:
```python
S = 12
latent_size = 15
device = 'cuda'
z0 = torch.randn(S,latent_size , device=device)
z1 = torch.randn(S, latent_size, device=device)
w = torch.linspace(0, 1, S, device=device).view(S, 1, 1)
z = (w * z0 + (1 - w) * z1).transpose(0, 1).reshape(S * S, latent_size)
```

```
x = vae_model.decoder(z)
show_images(x.data.cpu())
```



3 Conditional FC-VAE

The second model you'll develop will be very similar to the FC-VAE, but with a slight conditional twist to it. We'll use what we know about the labels of each MNIST image, and *condition* our latent space and image generation on the specific class. Instead of $q_\phi(z|x)$ and $p_\phi(x|z)$ we have $q_\phi(z|x, c)$ and $p_\phi(x|z, c)$

This will allow us to do some powerful conditional generation at inference time. We can specifically choose to generate more 1s, 2s, 9s, etc. instead of simply generating new digits randomly.

3.1 Define Network with class input (3 points)

Our CVAE architecture will be the same as our FC-VAE architecture, except we'll now add a one-hot label vector to both the x input (in our case, the flattened image dimensions) and the z latent space.

If our one-hot vector is called `c`, then `c[label] = 1` and `c = 0` elsewhere.

For the `CVAE` class in `vae.py` use the same FC-VAE architecture implemented in the last network with the following modifications:

1. Modify the first linear layer of your `encoder` to take in not only the flattened input image, but also the one-hot label vector `c`
2. Modify the first layer of your `decoder` to project the latent space + one-hot vector to the `hidden_dim`
3. Lastly, implement the `forward` pass to combine the flattened input image with the one-hot vectors (`torch.cat`) before passing them to the `encoder` and combine the latent space with the one-hot vectors (`torch.cat`) before passing them to the `decoder`

3.2 Train model

Using the same training script, let's now train our CVAE!

Training for 10 epochs should take ~2 minutes and your loss should be less than 120.

[15]:
```python
from vae import CVAE
num_epochs = 10
latent_size = 15
from nndl2.helper import train_vae
input_size = 28*28
device = 'cuda'

cvae = CVAE(input_size, latent_size=latent_size)
cvae.cuda()
for epoch in range(0, num_epochs):
  train_vae(epoch, cvae, loader_train, cond=True)
```

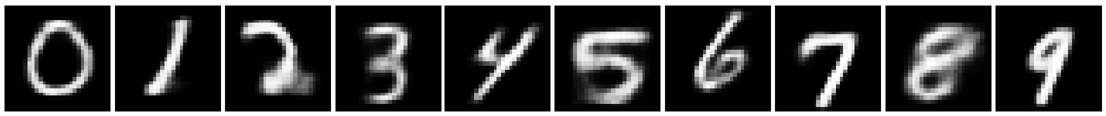
```
Train Epoch: 0  Loss: 137.802429
Train Epoch: 1  Loss: 131.535339
Train Epoch: 2  Loss: 124.594666
Train Epoch: 3  Loss: 115.604248
Train Epoch: 4  Loss: 118.044617
Train Epoch: 5  Loss: 110.885292
Train Epoch: 6  Loss: 114.613480
Train Epoch: 7  Loss: 108.608154
Train Epoch: 8  Loss: 107.402344
Train Epoch: 9  Loss: 106.561836
```

3.3 Visualize Results

We've trained our CVAE, now lets conditionally generate some new data! This time, we can specify the class we want to generate by adding our one hot matrix of class labels. We use `torch.eye` to create an identity matrix, gives effectively gives us one label for each digit. When you run the cell below, you should get one example per digit. Each digit should be reasonably distinguishable (it is ok to run this cell a few times to save your best results).

[16]:
```
z = torch.randn(10, latent_size)
c = torch.eye(10, 10) # [one hot labels for 0-9]
import matplotlib.gridspec as gridspec
z = torch.cat((z,c), dim=-1).to(device='cuda')
cvae.eval()
samples = cvae.decoder(z).data.cpu().numpy()

fig = plt.figure(figsize=(10, 1))
gspec = gridspec.GridSpec(1, 10)
gspec.update(wspace=0.05, hspace=0.05)
for i, sample in enumerate(samples):
  ax = plt.subplot(gspec[i])
  plt.axis('off')
  ax.set_xticklabels([])
  ax.set_yticklabels([])
  ax.set_aspect('equal')
  plt.imshow(sample.reshape(28, 28), cmap='Greys_r')
```



```python
from __future__ import print_function
import matplotlib.pyplot as plt
import matplotlib.gridspec as gridspec
import numpy as np
import torch
import torch.utils.data
from torch import nn, optim
from torch.autograd import Variable
from torch.nn import functional as F
from torchvision import datasets, transforms
from torchvision.utils import save_image


def hello_vae():
    print("Hello from vae.py!")


class VAE(nn.Module):
    def __init__(self, input_size, latent_size=15):
        super(VAE, self).__init__()
        self.input_size = input_size # H*W
        self.latent_size = latent_size # Z
        self.hidden_dim = None # H_d
        self.encoder = None
        self.mu_layer = None
        self.logvar_layer = None
        self.decoder = None

        ###########################################################################################
        # TODO: Implement the fully-connected encoder architecture described in the notebook.      #
        # Specifically, self.encoder should be a network that inputs a batch of input images of    #
        # shape (N, 1, H, W) into a batch of hidden features of shape (N, H_d). Set up             #
        # self.mu_layer and self.logvar_layer to be a pair of linear layers that map the hidden    #
        # features into estimates of the mean and log-variance of the posterior over the latent    #
        # vectors; the mean and log-variance estimates will both be tensors of shape (N, Z).       #
        ###########################################################################################
        # Replace "pass" statement with your code

        self.hidden_dim = 400

        # Define the encoder: A three Linear+ReLU layers neural network.
        self.encoder = nn.Sequential(
          nn.Flatten(),
          nn.Linear(self.input_size, self.hidden_dim),
          nn.ReLU(),
          nn.Linear(self.hidden_dim, self.hidden_dim),
          nn.ReLU(),
          nn.Linear(self.hidden_dim, self.hidden_dim),
          nn.ReLU()
        )

        # Define the mu_layer, with input (N, H_d) and output (N, Z)
        self.mu_layer = nn.Linear(self.hidden_dim, self.latent_size)
        # Define the logvar_layer, with input (N, H_d) and output (N, Z)
        self.logvar_layer = nn.Linear(self.hidden_dim, self.latent_size)

        ###########################################################################################
        # TODO: Implement the fully-connected decoder architecture described in the notebook.      #
        # Specifically, self.decoder should be a network that inputs a batch of latent vectors of  #
        # shape (N, Z) and outputs a tensor of estimated images of shape (N, 1, H, W).             #
        ###########################################################################################
        # Replace "pass" statement with your code

        self.decoder = nn.Sequential(
          nn.Linear(self.latent_size, self.hidden_dim),
          nn.ReLU(),
          nn.Linear(self.hidden_dim, self.hidden_dim),
          nn.ReLU(),
          nn.Linear(self.hidden_dim, self.hidden_dim),
          nn.ReLU(),
          nn.Linear(self.hidden_dim, self.input_size),
          nn.Sigmoid(),
          nn.Unflatten(dim=1, unflattened_size=(1, 28, 28))
        )
        ###########################################################################################
        #                                      END OF YOUR CODE                                    #
        ##########################################################################################


    def forward(self, x):
        """
        Performs forward pass through FC-VAE model by passing image through
        encoder, reparametrize trick, and decoder models

        Inputs:
        - x: Batch of input images of shape (N, 1, H, W)

        Returns:
        - x_hat: Reconstruced input data of shape (N,1,H,W)
        - mu: Matrix representing estimated posterior mu (N, Z), with Z latent space dimension
        - logvar: Matrix representing estimataed variance in log-space (N, Z), with Z latent space dimension
        """
        x_hat = None
        mu = None
        logvar = None
        ##########################################################################################
        # TODO: Implement the forward pass by following these steps                             #
        # (1) Pass the input batch through the encoder model to get posterior mu and logvariance  #
        # (2) Reparametrize to compute  the latent vector z                                      #
        # (3) Pass z through the decoder to resconstruct x                                       #
        ##########################################################################################
        # Replace "pass" statement with your code

        # Pass input images "x" to the encoder. Output shape is: (N, H_d)
        encoder_out = self.encoder(x)

        # Get the posterior mu from the encoder's output. Its shape is: (N, Z)
        mu = self.mu_layer(encoder_out)
        # Get the posterior logvariance from the encoder's output. Its shape is: (N, Z)
        logvar = self.logvar_layer(encoder_out)

        # Reparametrize to compute the latent vector "z", of shape (N, Z)
        z = reparametrize(mu, logvar)

        # Pass "z" through the decoder to resconstruct "x", the "x_hat".
        x_hat = self.decoder(z)

        ##########################################################################################
        #                                       END OF YOUR CODE                                 #
        ##########################################################################################
        return x_hat, mu, logvar


class CVAE(nn.Module):
    def __init__(self, input_size, num_classes=10, latent_size=15):
        super(CVAE, self).__init__()
        self.input_size = input_size # H*W
        self.latent_size = latent_size # Z
        self.num_classes = num_classes # C
        self.hidden_dim = None # H_d
        self.encoder = None
        self.mu_layer = None
        self.logvar_layer = None
        self.decoder = None

        ##########################################################################################
        # TODO: Define a FC encoder as described in the notebook that transforms the image--after  #
        # flattening and now adding our one-hot class vector (N, H*W + C)--into a hidden_dimension #                #
        # (N, H_d) feature space, and a final two layers that project that feature space           #
        # to posterior mu and posterior log-variance estimates of the latent space (N, Z)          #
        ##########################################################################################
        # Replace "pass" statement with your code

        self.hidden_dim = 400

        # Define the encoder: A three Linear+ReLU layers neural network.
        # The encoder's input has shape of (N, H*W + C), i.e. The flattened images and their classes.
        # Note that the concatenation (between images and classes) is done in the "forward" function.
        self.encoder = nn.Sequential(
          nn.Linear(self.input_size + self.num_classes, self.hidden_dim),
          nn.ReLU(),
          nn.Linear(self.hidden_dim, self.hidden_dim),
          nn.ReLU(),
          nn.Linear(self.hidden_dim, self.hidden_dim),
          nn.ReLU()
        )

        # Define the mu_layer, with input (N, H_d) and output (N, Z)
        self.mu_layer = nn.Linear(self.hidden_dim, self.latent_size)
        # Define the logvar_layer, with input (N, H_d) and output (N, Z)
        self.logvar_layer = nn.Linear(self.hidden_dim, self.latent_size)

        ##########################################################################################
        # TODO: Define a fully-connected decoder as described in the notebook that transforms the  #
        # latent space (N, Z + C) to the estimated images of shape (N, 1, H, W).                   #
        ##########################################################################################
        # Replace "pass" statement with your code

        self.decoder = nn.Sequential(
          nn.Linear(self.latent_size + self.num_classes, self.hidden_dim),
          nn.ReLU(),
          nn.Linear(self.hidden_dim, self.hidden_dim),
          nn.ReLU(),
          nn.Linear(self.hidden_dim, self.hidden_dim),
          nn.ReLU(),
          nn.Linear(self.hidden_dim, self.input_size),
          nn.Sigmoid(),
          nn.Unflatten(dim=1, unflattened_size=(1, 28, 28))
        )

        ##########################################################################################
        #                                      END OF YOUR CODE                                    #
        ##########################################################################################

    def forward(self, x, c):
        """
        Performs forward pass through FC-CVAE model by passing image through
        encoder, reparametrize trick, and decoder models

        Inputs:
        - x: Input data for this timestep of shape (N, 1, H, W)
        - c: One hot vector representing the input class (0-9) (N, C)

        Returns:
        - x_hat: Reconstruced input data of shape (N, 1, H, W)
        - mu: Matrix representing estimated posterior mu (N, Z), with Z latent space dimension
        - logvar: Matrix representing estimated variance in log-space (N, Z),  with Z latent space dimension
        """
        x_hat = None
        mu = None
        logvar = None
        ##########################################################################################
        # TODO: Implement the forward pass by following these steps                                #
        # (1) Pass the concatenation of input batch and one hot vectors through the encoder model  #
        # to get posterior mu and logvariance                                                      #
        # (2) Reparametrize to compute the latent vector z                                         #
        # (3) Pass concatenation of z and one hot vectors through the decoder to resconstruct x    #
        ##########################################################################################
        # Replace "pass" statement with your code

        # Flatten the "height" and "width" of the input batch 'x'.
        # Input shape (N, 1, H, W). Output shape: (N, H*W)
        x_flat = torch.flatten(x, start_dim=1, end_dim=-1)

        # Create the encoder's input 'enc_in' by concatenating flattened images and their classes.
        # Output shape is: (N, H*W + C)
        enc_in = torch.cat((x_flat, c), dim=1)

        # Pass 'enc_in' to the encoder. Output shape is: (N, H_d)
        enc_out = self.encoder(enc_in)

        # Get the posterior mu from the encoder's output. Its shape is: (N, Z)
        mu = self.mu_layer(enc_out)
        # Get the posterior logvariance from the encoder's output. Its shape is: (N, Z)
        logvar = self.logvar_layer(enc_out)

        # Reparametrize to compute the latent vector "z", of shape (N, Z)
        z = reparametrize(mu, logvar)
```

```
        # Create the decoder's input 'dec_in' by concatenating the latent vector and its classes.
        # Output shape is: (N, Z + C)
        dec_in = torch.cat((z, c), dim=1)

        # Pass "dec_in" through the decoder to resconstruct "x", the "x_hat".
        # x.shape == x_hat.shape == (N, 1, H, W)
        x_hat = self.decoder(dec_in)

        ###############################################################################################
        #                                      END OF YOUR CODE                                      #
        ###############################################################################################
        return x_hat, mu, logvar



def reparametrize(mu, logvar):
    """
    Differentiably sample random Gaussian data with specified mean and variance using the
    reparameterization trick.

    Suppose we want to sample a random number z from a Gaussian distribution with mean mu and
    standard deviation sigma, such that we can backpropagate from the z back to mu and sigma.
    We can achieve this by first sampling a random value epsilon from a standard Gaussian
    distribution with zero mean and unit variance, then setting z = sigma * epsilon + mu.

    For more stable training when integrating this function into a neural network, it helps to
    pass this function the log of the variance of the distribution from which to sample, rather
    than specifying the standard deviation directly.

    Inputs:
    - mu: Tensor of shape (N, Z) giving means
    - logvar: Tensor of shape (N, Z) giving log-variances

    Returns:
    - z: Estimated latent vectors, where z[i, j] is a random value sampled from a Gaussian with
         mean mu[i, j] and log-variance logvar[i, j].
    """
    z = None
    ###############################################################################################
    # TODO: Reparametrize by initializing epsilon as a normal distribution and scaling by        #
    # posterior mu and sigma to estimate z                                                        #
    ###############################################################################################
    # Replace "pass" statement with your code

    # Convert the "log of the variance" to "sigma" (standard deviation).
    sigma = torch.sqrt(torch.exp(logvar))

    # Compute 'z'.
    # Epsilon is a Tensor that contains random samples from a standard normal
    # distribution (mu=0, std=1)
    z = sigma * torch.randn_like(mu) + mu

    ###############################################################################################
    #                                END OF YOUR CODE                                            #
    ###############################################################################################
    return z


def loss_function(x_hat, x, mu, logvar):
    """
    Computes the negative variational lower bound loss term of the VAE (refer to formulation in notebook).

    Inputs:
    - x_hat: Reconstruced input data of shape (N, 1, H, W)
    - x: Input data for this timestep of shape (N, 1, H, W)
    - mu: Matrix representing estimated posterior mu (N, Z), with Z latent space dimension
    - logvar: Matrix representing estimated variance in log-space (N, Z), with Z latent space dimension

    Returns:
    - loss: Tensor containing the scalar loss for the negative variational lowerbound
    """
    loss = None
    ###############################################################################################
    # TODO: Compute negative variational lowerbound loss as described in the notebook            #
    ###############################################################################################
    # Replace "pass" statement with your code

    # Get the minibatch size
    N = mu.shape[0]

    # Compute the reconstruction loss term, using Binary Cross Entropy (BCE) loss.
    # The "BCE loss" have to be adapted to the "reconstruction loss" (Expectation) by:
    # - Changing the reduction mode from 'mean' (default) to 'sum' (used in the Expectation).
    # - The input to the BCE is 'x_hat' and the target is 'x'. This can be done because we are
    # operating on MNIST dataset, where each pixel is either 0 or 1.
    # Note that the minus sign is handled by the BCE loss itself.
    rec_term = nn.functional.binary_cross_entropy(x_hat, x, reduction='sum')

    # Compute the KL divergence term (kldiv_term).
    kldiv_term = 1 + logvar - mu**2 - torch.exp(logvar)
    kldiv_term = -0.5 * kldiv_term.sum()

    # Final loss is the sum of "reconstruction loss term" and "KL divergence term".
    loss = rec_term + kldiv_term

    # Average the loss across samples in the minibatch.
    loss /= N


    ################################################################################################
    #                            END OF YOUR CODE                                                  #
    ################################################################################################
    return loss
```